Continuous Control Project Report

Description of implementation:

The goal of this project was to train an agent to solve the Unity Reacher environment. The solution requires moving the double-jointed reacher arm to a moving target location, and remaining in the target location for as long as possible. As this is a reinforcement learning task, a reward of +0.1 is provided for each step that the reacher's "hand" is in the target location. For this implementation, the 20-agent (rather than the single agent) version of the environment was used. This version has 20 agents (reacher arm copies) operating simultaneously. For each agent, the observation space consists of 33 variables corresponding to position, rotation, velocity, and angular velocities of the arm. Each action is a vector with four numbers, corresponding to torque applicable to two joints. Every entry in the action vector should be a number between -1 and 1.

This task requires continuous control of the reacher arm. Thus, algorithms such as Deep Q-Network (DQN), that work well in discrete action spaces, are not applicable to this environment. Instead, an algorithm known as Deep Deterministic Policy Gradients (DDPG) was employed. This algorithm was designed to work with continuous action spaces. DDPG is a type of actor-critic algorithm. The actor seeks out the optimal policy by taking a policy gradient approach. The actor is aided by the critic, which in this case tries to find the optimal Q function. Both the actor and critic each have their own neural network, which are implemented using pytorch.

The code for this project was adapted from the DDPG code that was written to operate on OpenAI Gym's Pendulum environment. The code includes some techniques that were recently developed in the field of deep reinforcement learning. One of these techniques is known as "Experience Replay", in which data returned from the environment (experiences) is stored in a "replay buffer". A batch of these experiences is then chosen randomly from the replay buffer to train the network. This technique insures that the experiences are uncorrelated (correlated data would tend to destabilize training). Another technique used here is "Soft Update for Target Networks". This technique allows the target networks (actor and critic targets) to be updated gradually, rather than making sudden all-at-once updates. This technique also helps to stabilize learning. Exploration is facilitated by adding noise to the actions.

Hyperparameters for Continuous Control Notebook:

n_episodes = 2500 (maximum number of episodes)

print_every = 10 (how often to print episode number and average score)

Hyperparameters for ddpg_agent:

BUFFER_SIZE = int(1e5) (replay buffer size)
BATCH_SIZE = 128 (minibatch size)
GAMMA = 0.99 (discount factor)
TAU = 1e-3 (for soft update of target parameters)
LR_ACTOR = 1e-4 (actor learning rate)
LR_CRITIC = 1e-3 (critic learning rate)
WEIGHT_DECAY = 0 (L2 weight decay)

Hyperparameters for model (pytorch network):

action_size = 4 (action size is fixed by the environment)
state_size = 33 (state size is fixed by the environment)
Number of nodes in first layer of actor network set to 400
Number of nodes in second layer of actor network set to 300
Number of nodes in first layer of critic network set to 400
Number of nodes in second layer of critic network set to 300

Suggestions for Improvement:

The algorithm was successful in solving the environment, achieving an average score of 30 after 277 episodes in the final run. However, learning appeared to be somewhat unstable, as a few “random restarts” were required to achieve success. For future implementations, some stabilization techniques could be added. These include gradient-clipping (during training of the critic network) and adjusting the frequency of actor and critic network updates, as suggested in the “Benchmark Implementation” section of the lesson.